Execution of Graph Algorithms on GPU Graph Frameworks

- *By Bhaskar Khaneja*

Abstract

Graphical Processing Units (GPUs) are well-suited for graph analytics problems because of their ability to parallelize applications and speed up computation. As a result, over the last decade, several high-level GPU graph frameworks have evolved [9]; these frameworks allow programmers to concentrate on expressing primitives, since they themselves take care of scaling up to parallel architectures. A few examples of these frameworks include nvGRAPH [5], CuSha [3] and Gunrock [8]. Each one of these frameworks is built around a unique programming model, which governs how it manages data movement, memory accesses, load balancing as well as how it maps irregular graph topologies to parallel hardware.

While there have been studies of individual high-level GPU graph frameworks in the past [3, 5, 8], a comprehensive study comparing their differences has not been done. A thorough investigation of these frameworks' performance can provide useful insights about the benefits of using specific programming models for specific graph topologies, and help accelerate a variety of irregular graph applications.

In this research, we execute two graph algorithms – Single Source Shortest Paths (SSSP) and PageRank – on three high-level GPU graph frameworks – nvGRAPH, CuSha, and Gunrock – on a set of diverse graph topologies, and benchmark performance on NVIDIA Tesla P40 and P100 GPUs.

The results show that CuSha provides the best-in-class performance on traversal-based primitives such as SSSP, whereas nvGRAPH and Gunrock are the best frameworks to use for dense-based-computation primitives such as PageRank. Between nvGRAPH and Gunrock for PageRank, nvGRAPH is better suited for large low-diameter graphs, Gunrock is better suited for large high-diameter graphs, and they are both equally well-suited for small graphs of any diameter.

Introduction

Graphs are a crucial tool for analyzing relationships between entities in a wide variety of computational fields. By representing entities as nodes and the relationships between entities as edges, graphs greatly simplify real-world problems and provide a useful abstraction for solving them. Many graphs found in real-world applications are huge and complex, and effectively making real-time decisions based on them requires incredible graph processing speed. This is where Graphics Processing Units (GPUs) come in handy. GPUs are power-efficient and high-memory-bandwidth processors that exploit parallelism in data-intensive applications to speed up computation [9].

In the past decade, a number of shared or distributed memory GPU graph frameworks have emerged. These frameworks allow programmers to easily run graph primitives on GPUs by letting them focus only on the primitives' expression and themselves taking care of all the automatic scaling up of computation to parallel architectures. Examples of such frameworks include nvGRAPH [5], CuSha [3] and Gunrock [8]. Each one of these frameworks is built on top of a unique programming model [9]. For instance, nvGRAPH models graph problems as linear algebra problems and uses Sparse Matrix Vector Product (SPMV) with a semi-ring model and automatic load balancing for sparsity patterns to handle graph analytics problems [5]. On the other hand, CuSha employs a Gather-Apply-Scatter (GAS) approach, where it uses special data structures called G-Shards and Concatenated Windows (CW) to iteratively apply a compute function to every vertex in the graph until a convergence condition is met [3]. Lastly, Gunrock uses a data-centric

model focused on operations of a subset of vertices and/or edges, where it dynamically chooses optimization strategies during runtime based on graph topology [8].

Despite the development of high-level GPU graph frameworks, the focus of research in the field of graph processing has still been the architectural-level characterization of GPU graph primitives. For instance, researchers have identified frequent kernel calls and ineffective use of caches to be the biggest bottlenecks limiting GPU performance [9]. Through simulation-based analysis, researchers have also studied the low-level behavior of Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Threads (SIMT) architectures, control-flow and memory-access irregularities, as well as the underutilization of GPU execution cycles due to irregular graph codes [9]. However, the addition of high-level abstractions for graph analytics GPU-based frameworks and their impact on performance has not been investigated in detail. It remains unclear how to map the low-level optimizations and performance bottlenecks to high-level design choices and programming models for optimal performance on graph analytics problems. Given the fundamental differences in GPU graph frameworks' programming models, it is possible that a graph algorithm executed on one framework might yield better performance than on the other. Moreover, a framework's performance using a specific graph algorithm may also be dependent on the structure of the graph used. In this research, we attempt to address these questions by executing two graph primitives – Single Source Shortest Paths (SSSP) and PageRank – on nvGRAPH, CuSha and Gunrock GPU graph frameworks on a set of structurally different graphs, and benchmark performance.

Background

Modern applications create large-scale graph structures with billions and trillions of vertices and edges. The need to quickly analyze them to glean useful insights and then to use those insights to make better, more informed decisions has resulted in an increase in the popularity of graph analytics. Today, graph analytics is used in a variety of commercial applications across domains such as genomics, transportation, biological sciences, finance and engineering.

At the heart of graph analytics lie special devices called Graphics Processing Units (GPUs). GPUs are power-efficient and high-memory-bandwidth microprocessors that are extremely good as accelerators; they derive their power from a massively parallel architecture and the ability to exhibit regular memory access patterns, few synchronizations and a direct mapping to parallel hardware [2]. To allow developers to tap into GPUs' monstrous power, over the years, several high-level programmable GPU graph frameworks have been written. These frameworks let developers implement various types of complex graph applications called graph primitives, without worrying about the inner details related to parallelization and synchronization [9]. While these frameworks all provide similar flexibility in the kinds of GPU programs which can be written on them, they are each based on a different programming model [9].

1. nvGRAPH *(Linear Algebra Model)*

   nvGRAPH is a close-sourced high-performance GPU graph analytics library developed by NVIDIA. It expresses graph analytics problems as linear algebra and matrix computations,

and uses semi-ring Sparse Matrix Vector Product (SPMV) to perform graph operations [5]. The library currently supports three algorithms: PageRank, Single Source Shortest Path (SSSP), and Single Source Widest Path (SSWP).

2. CuSha *(Gather-Apply-Scatter Model)*

CuSha is an open-sourced CUDA-based graph processing framework based on the Gather-Apply-Scatter (GAS) model [3]. The GAS model decomposes a vertex program into three conceptual phases: *gather*, *apply*, and *scatter* [3]. The *gather* phase accumulates information about adjacent vertices and edges of each active vertex through a generalized binary operation over its neighbor list [9]. The *apply* phase executes the accumulated value, the output of the gather phase, to the active vertices until convergence [9]. Finally, the *scatter* phase evaluates a predicate on all adjacent vertices and broadcasts the result along outgoing edges [9]. Internally, CuSha makes use of two data structures called G-Shards and Concatenated Windows (CW) [3]. G-Shards distributes graph data in a manner that places edges and vertices required by a subset of computation contiguously in memory, thereby providing better memory locality than the Compressed Sparse Row (CSR) graph representation [3]. Concatenated Windows (CW) on the other hand, concatenates multiple computation windows from shards so that the GPU threads are always highly utilized [3]. CuSha also provides asynchronous execution that makes updated values visible in the same iteration, thereby resulting in faster convergence [3].

3. Gunrock *(Data-Centric Model)*

Gunrock is an open-sourced CUDA-based graph processing framework based on the data-centric model [8]. Its abstraction works by manipulating sets of vertices and edges (called frontiers) that are actively participating in the computation instead of focusing on expressing sequential steps of computation on vertices or edges as in the case of vertex- and edge-centric models [8]. The library supports three ways of manipulating the current frontier: 1) *advance* generates a new frontier by visiting the neighbors of the current vertex frontier, 2) *filter* generates a new frontier by choosing a subset of the current frontier based on the programmer's specifications, and 3) *compute* performs an operation on all vertices of the current frontier in parallel [8]. Gunrock targets graph operations expressible as iterative convergent processes and unlike CuSha, supports both vertex and edge frontiers [8]. Gunrock also integrates complex coarse- and fine-grained load-balancing and work-efficiency strategies into its core [8], making it relatively simpler for programmers to implement high performing graph analytics.

In addition, CuSha and Gunrock also employ the Bulk-Synchronous Parallel (BSP) paradigm, in which a graph problem is executed as a series of consecutive "super-steps", separated by global-barriers, where each super-step involves data-parallelism over the vertices and edges in a frontier, and is executed efficiently on a GPU [9].

Related Work

A significant part of research in the field of graph analytics has focused on developing and evaluating graph primitives, in order to identify bottlenecks that limit GPU performance on graph problems. Researchers have found common performance deterrents to include an excessive number of kernel invocations, ineffective use of caches, underutilized execution cycles, branch divergence, load imbalance, synchronization overhead, memory coalescing, L2/DRAM latency and DRAM bandwidth [9]. To dissect this idea further, other researchers have characterized low-level behavior of Single Instruction Multiple Data (SIMD) architectures, including cache hit ratios, execution time breakdown, speedups over CPU version execution, and Single Instruction Multiple Threads (SIMT) lane utilization [9].

While GPUs' architectural-level behavior has been investigated in detail to identify potential sources of performance gains and losses on graph analytics problems, not a lot of work has gone into comparing the high-level GPU graph frameworks and how their respective programming models might impact performance. Given the vast differences in the frameworks' programming models, it is possible that a specific combination of graph primitive and programming model might be better suited to one kind of graph topology and not the other. The objective of this research is to further explore this idea by executing two graph primitives – Single Source Shortest Path (SSSP) and PageRank – on three GPU graph frameworks – nvGRAPH, CuSha and Gunrock – and benchmarking performance.

Methodology

nvGRAPH, CuSha and Gunrock are three GPU graph frameworks based on linear algebra, GAS and data-centric programming models respectively. Our goal is to benchmark the performance of select graph primitives on select graph topologies using these libraries, in order to get a better sense of which combinations of graph primitives and programming models work best for which graph topologies.

Selection of Graph Primitives

Graph primitives can roughly be split into two broad classes – *traversal-based* primitives and *dense-computation-based* primitives, based on the activity of the vertices involved [9]. *Traversal-based* primitives start from a subset of vertices in the graph and systematically explore and update neighboring vertices until all reachable vertices have been visited [9]. At any point in time during computation, only a subset of vertices is active. On the other hand, in the case of *dense-computation-based* primitives, most or all vertices are active throughout the computation period [9]. In this research, we use two graph primitives – Single Source Shortest Path (SSSP) and PageRank – as case-studies to benchmark the performance of GPU graph frameworks. These graph primitives are chosen because they together span both memory- and computation-bound behaviors and broadly reflect a typical workload in graph analytics. For example, SSSP is a *traversal-based* graph primitive that includes expensive computations and complicated traversal paths [1]. PageRank on the contrary is a *dense-computation-based* primitive which involves dense computations with different per vertex/edge workloads [9]. The graph primitives are briefly described below:

1. Single Source Shortest Path (SSSP)

   Given a graph with edge weights and a source vertex, SSSP computes the shortest distances between the source vertex and all other vertices in the graph.

2. PageRank

   PageRank is a link analysis algorithm that assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of quantifying its relative importance in the set. This is the algorithm that Google Search uses to rank websites in its search engine results [12]. The iterative method of computing PageRank gives each vertex an initial value and updates it based on the PageRank of its neighbors, until the PageRank value for each vertex converges [12]. Mathematically,

$$PR(p_i) = (1 - d) + d \sum_{p_j \varepsilon M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

   where p_i is the page under consideration, d is the damping factor, $M(p_i)$ is the set of pages that link to p_i, $L(p_j)$ is the number of outbound links on page p_j, and N is the total number of pages [12]. d or the damping factor, denotes the probability that a random surfer on the web continues clicking on links without stopping, and is generally assumed to be 0.85 [12].

Selection of Graph Topologies

Real-world graph topologies can also be divided into two classes: one containing low diameter graphs with highly skewed scale-free degree distributions and the other containing large diameters with evenly-distributed degrees [9]. The average number of degrees of a graph and the degree distribution can greatly impact the amount of parallelism, and so for this research we pick diverse datasets that encompass both categories. Specifically, the graph topologies used have been pulled from two different public repositories – Stanford Network Analysis Project (SNAP) [6], and Suite Sparse Matrix Collection [7]. Low diameter graphs include *orkut*, *email-Enron*, *cage15*, *soc-Slashdot0902*, *indochina-2004*, *kron_g500-logn21*, and *wb-edu*, and high diameter graphs include *dblp*, *com-amazon*, *amazon0601*, *cit-Patents*, *liveJournal*, and *soc-LiveJournal1*.

Name	Type	Nodes	Edges	90th Percentile Effective Diameter	Description	Source
email-Enron	Undirected	36,692	183,831	4.8	Email communication network from Enron	[6]
com-DBLP	Undirected, Communities	317,080	1,049,866	8	DBLP collaboration network	[6]
com-amazon	Undirected, Communities	334,863	925,872	15	Amazon product network	[6]
amazon0601	Directed	403,394	3,387,388	7.6	Amazon product co-purchasing network from June 1 2003	[6]
Skitter	Undirected	1,696,415	11,095,298	6	Internet topology graph, from traceroutes run daily in 2005	[6]
cit-Patents	Directed, Temporal, Labeled	3,774,768	16,518,948	9.4	Citation network among US Patents	[6]
LiveJournal	Undirected, Communities	3,997,962	34,681,189	6.5	LiveJournal online social network	[6]
soc-LiveJournal1	Directed	4,847,571	68,993,773	6.5	LiveJournal online social network	[6]
com-Orkut	Undirected, Communities	3,072,441	117,185,083	4.8	Orkut online social network	[6]
wikipedia-2007	Directed	3,566,907	45,030,389	-	-	[7]
wb-edu	Directed	9,845,725	99,199,551	-	-	[7]
cage15	Directed Weighted	5,154,859	99,199,551	-	-	[7]
kron_g500_logn21	Undirected Multigraph	2,097,152	182,082,942	-	-	[7]
soc-Slashdot0902	Directed	82,168	948,464	-	-	[7]
indochina-2004	Directed	7,414,866	194,109,311	-	-	[7]

Table 1: Metadata for graphs used in experiments

Experiments & Results

To benchmark the performance of nvGRAPH, CuSha and Gunrock GPU graph frameworks on SSSP and PageRank graph primitives, experiments were run on the NVIDIA Tesla P40 and NVIDIA Tesla P100 accelerators on a system running Linux with an Intel(R) Xeon(R) CPU E5-2695 v4. The NVIDIA Tesla P40 is powered by the NVIDIA Pascal architecture, has 24 GB of GDDR5 memory, 30 Streaming Multiprocessors (SMs), and 3,840 CUDA cores [13]. The NVIDIA Tesla P100 is also based on the NVIDIA Pascal architecture, but unlike the P40, has 16 GB of HBM2 memory, 60 SMs, and 3,584 CUDA cores [14]. Both GPUs have similar architectures, but are quite different in their memory subsystems. To better understand these differences and their impact on performance, all experiments were performed on both GPUs. In the following sections, we report the results from both the P40 and the P100, but do not get into the analysis of their differences due to the complex nature of the subject. Instead, we focus our discussion on the evaluation of the behavior of high-level GPU graph frameworks on different graph topologies.

For every GPU graph framework, the graph topologies were fed as input in either the Coordinate (COO) or the Compressed Sparse Column (CSC) format. Since the focus of this research was to gain an abstraction-level understanding of the GPU frameworks, the times taken to copy data from host to device and device to host were disregarded. Speedup was computed by dividing the processing time on P100 by the processing time on P100 or P40. For all the SSSP experiments, 200 different source vertices were selected and the average processing time was reported in seconds. Similarly, for all the PageRank experiments, 5 iterations were performed with

an initial guess of 0 and damping factor of 0.85, and the times were once again averaged to obtain the average processing time per iteration in seconds.

1. nvGRAPH

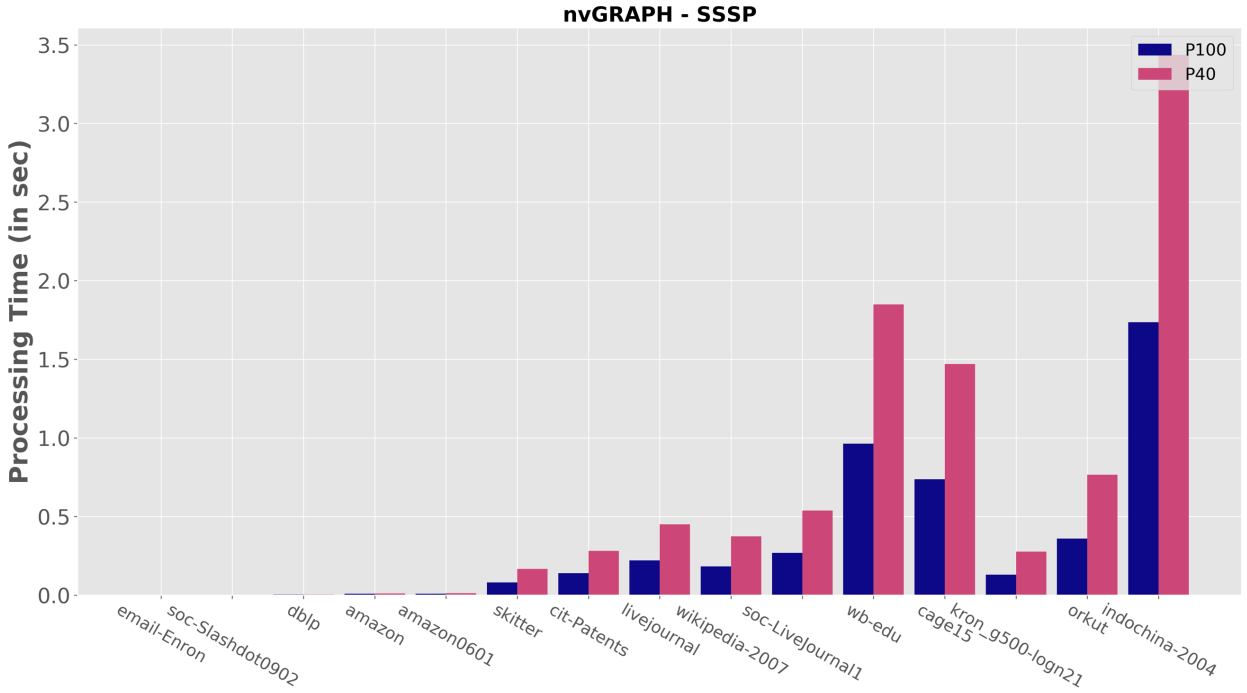

Figure 1: nvGRAPH SSSP processing times on NVIDIA Tesla P100 and P40 GPUs

On both the NVIDIA Tesla P40 and P100, on average, processing SSSP took the longest on *indochina-2004* at ~3.4 sec (P40) and ~1.75 sec (P100) followed by *wb-edu* at ~1.8 sec (P40) and 0.9 sec (P100) and *cage15* at ~1.5 sec (P40) and 0.75 sec (P100). All the other graph topologies, no matter how big or small they were, took under 1 sec (P40) and 0.4 sec (P100) to process.

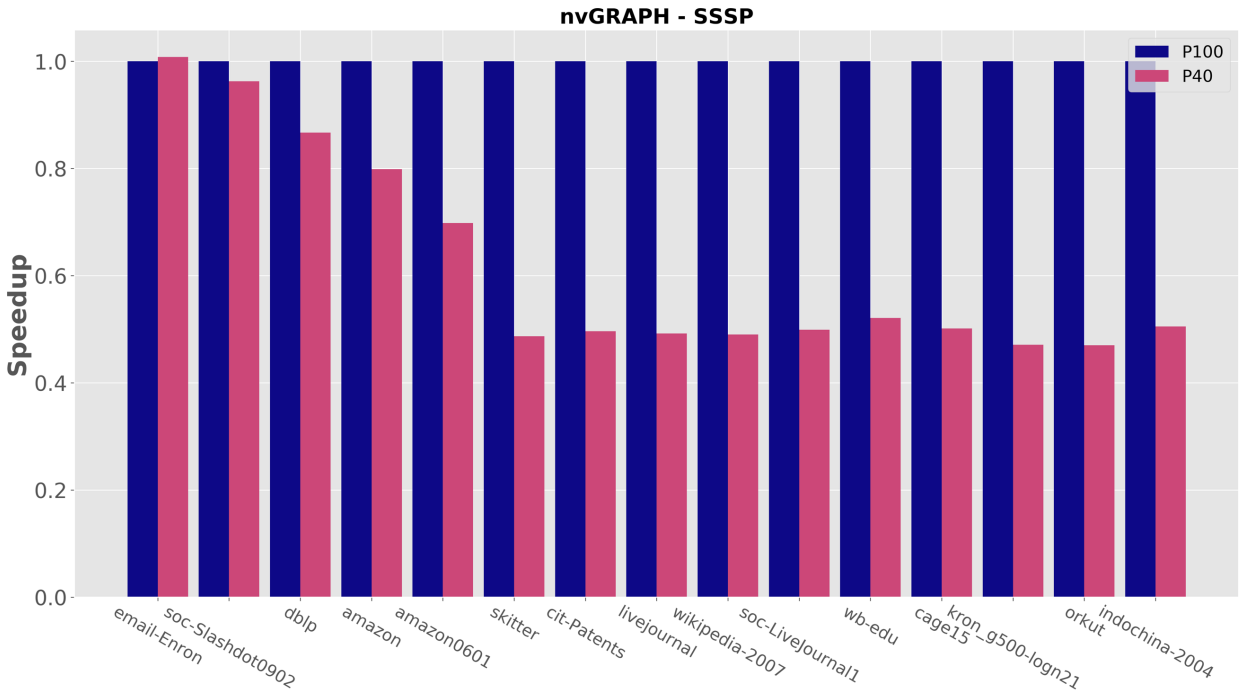

Figure 2: nvGRAPH SSSP speedup - NVIDIA Tesla P100 vs P40

With SSSP on nvGRAPH, NVIDIA Tesla P100 performed better than NVIDIA Tesla P40 on all graphs, except for *email-Enron*, where it was slightly worse. On the majority of mid- and large-sized graphs, P100 was almost twice as fast as P40.

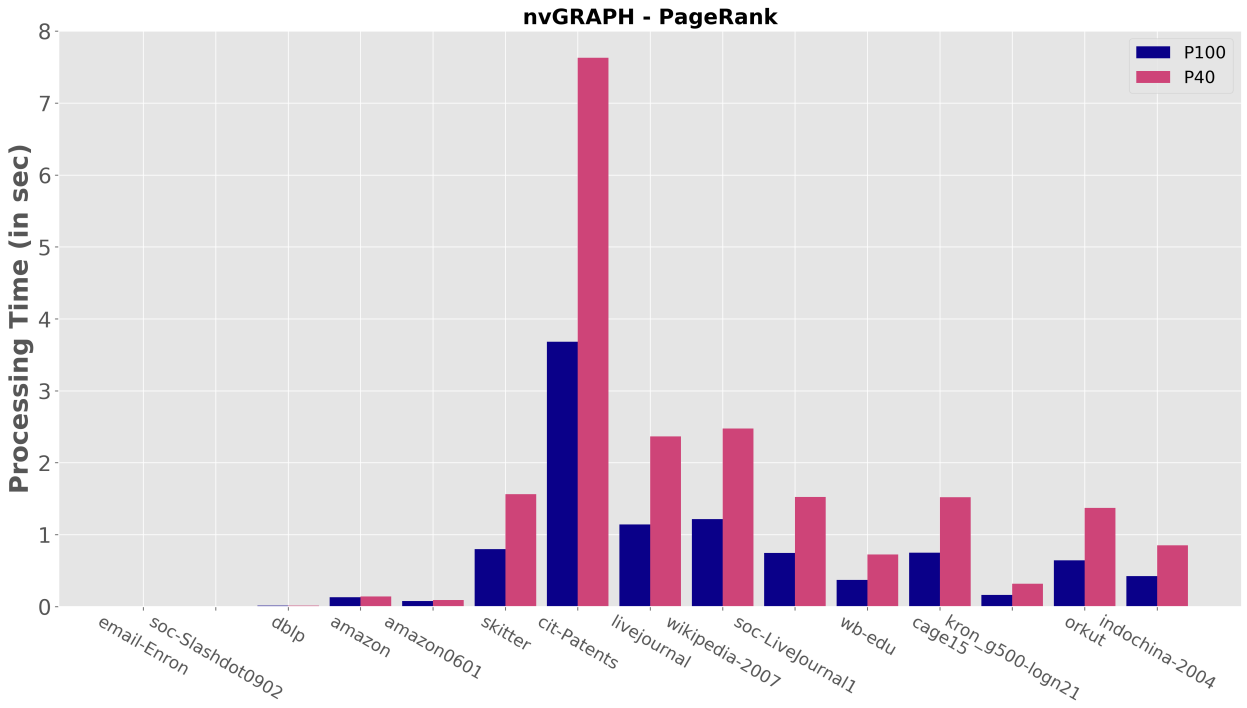

Figure 3: nvGRAPH PageRank processing times on NVIDIA Tesla P100 & P40 GPUs

In the case of PageRank, both NVIDIA Tesla P40 and P100 exhibited similar behaviors also. Processing *cit-Patents* took the longest at ~7.6 sec (P40) and ~3.7 sec (P100) followed by a close match between *livejournal* and *wikipedia-2007* at ~2.4 sec (P40) and ~1.25 sec (P100), followed by all the other graph topologies which took less than 2 sec (P40) and 0.8 sec (P100).

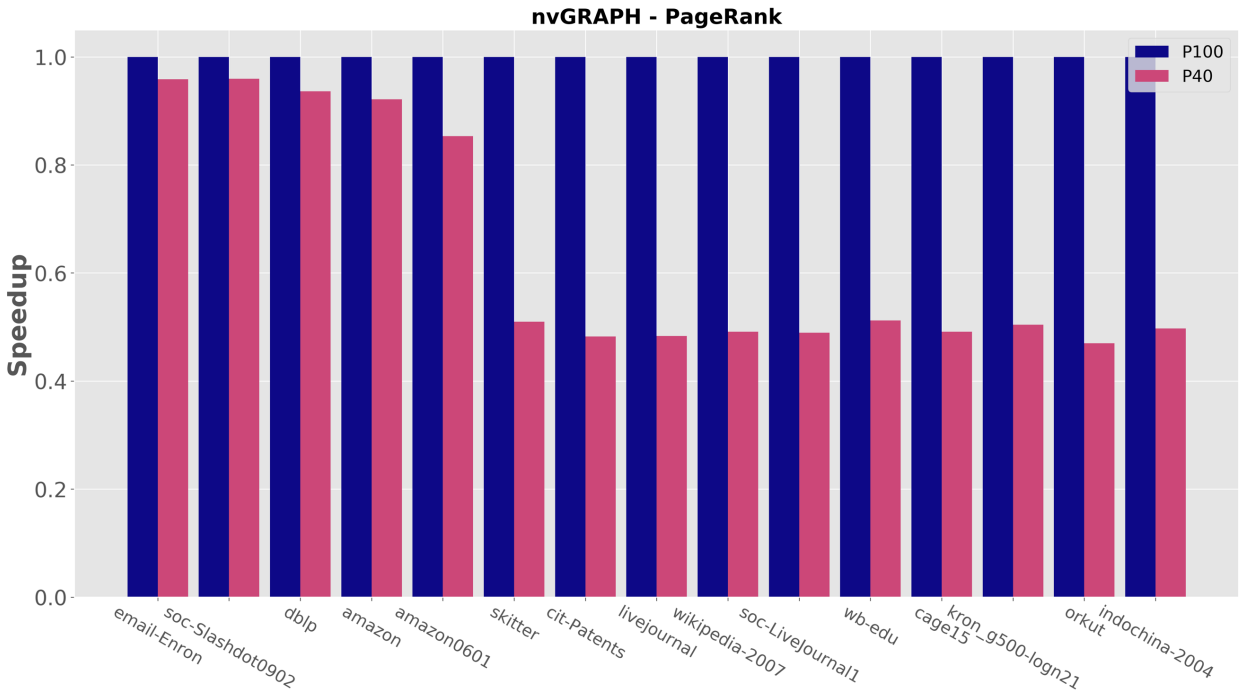

Figure 4: nvGRAPH PageRank speedup - NVIDIA Tesla P100 vs P40

Executing PageRank on nvGRAPH on NVIDIA Tesla P100 yielded better performance than on NVIDIA Tesla P40 on all graphs. Once again, P100 was twice as fast as P40 on most of mid- and large-sized graphs.

2. CuSha

CuSha experiments were performed with both Concatenated Windows (CW) and G-Shard (GS) representations as methods.

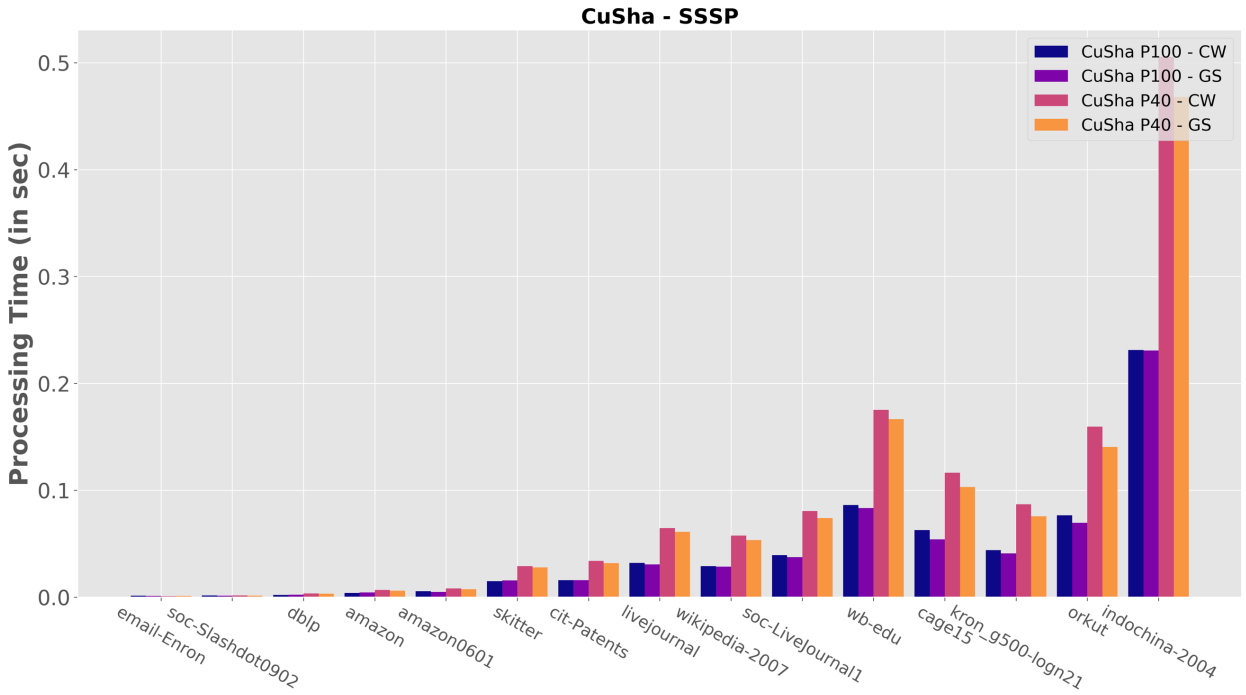

Figure 5: CuSha SSSP processing times on NVIDIA Tesla P100 and P40 GPUs

The relative performances on both the NVIDIA Tesla P40 and P100 were also similar for SSSP on CuSha. Processing *indochina-2004* took the longest at ~0.45 to ~0.51 sec (P40) and ~0.23 sec (P100). This was followed by *wb-edu* and *orkut* at ~0.17 to ~0.18 sec (P40) and ~0.07 to ~0.08 sec (P100) respectively, followed by all the other graphs. Between CW and GS, the performance was slightly better with GS on all graphs.

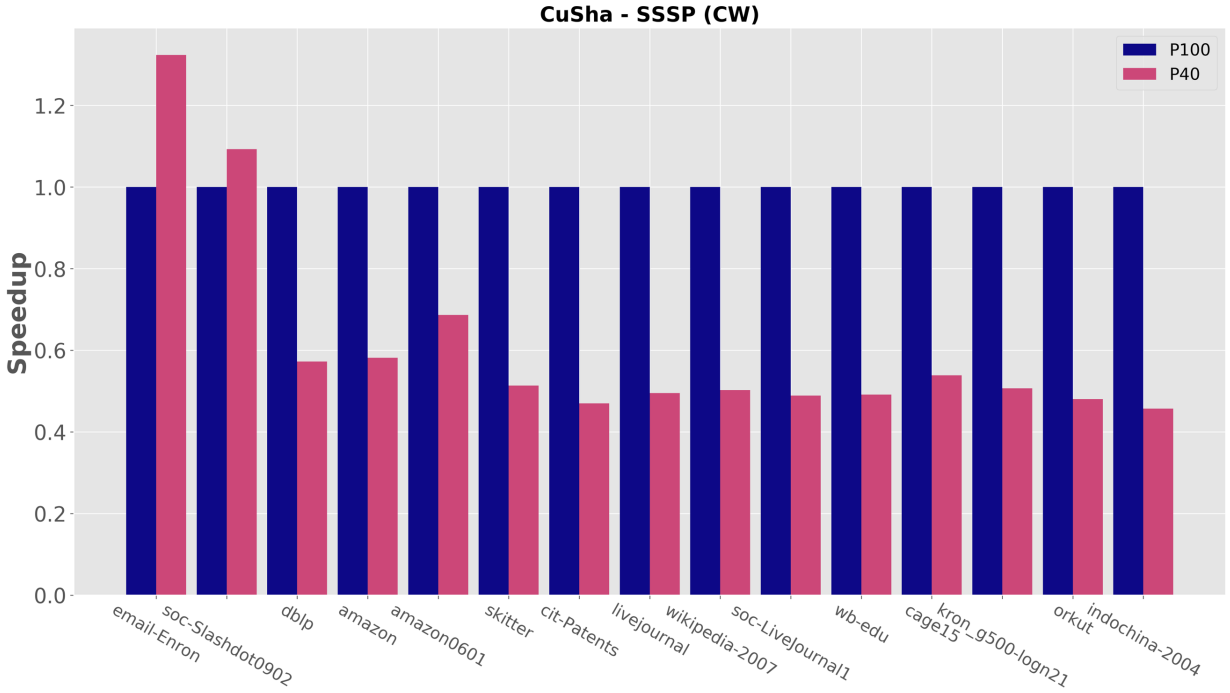

Figure 6: CuSha SSSP (CW) speedup - NVIDIA Tesla P100 vs P40

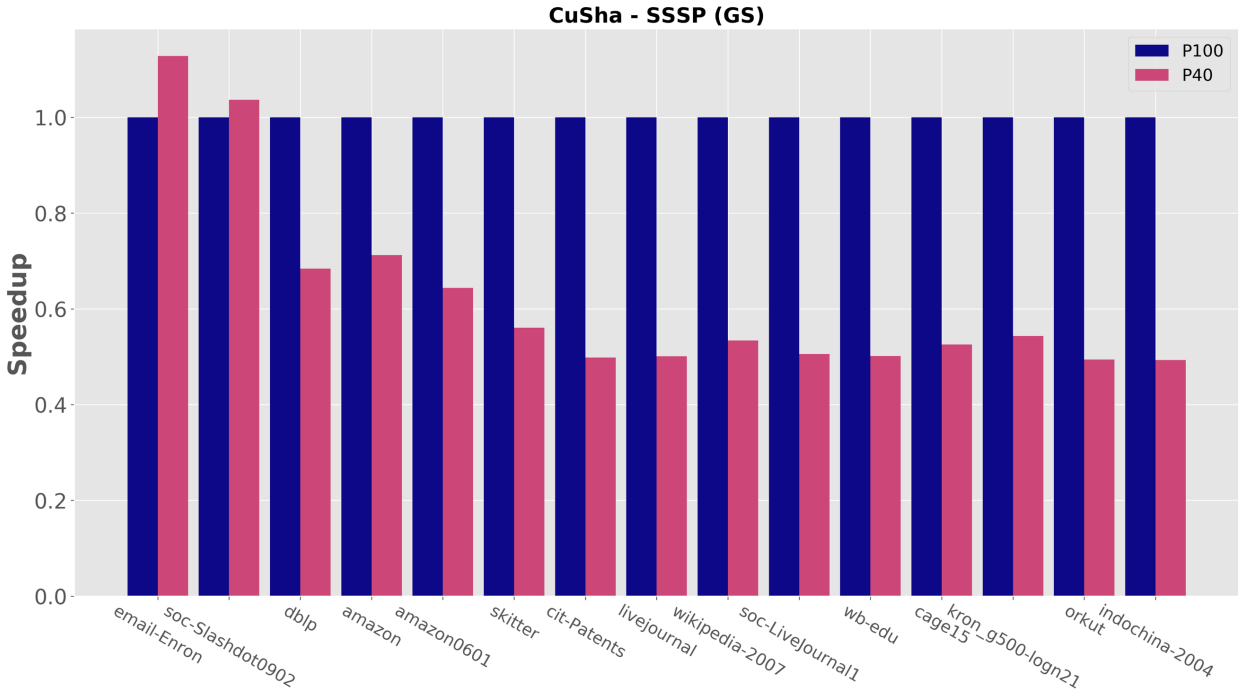

Figure 7: CuSha SSSP (GS) speedup - NVIDIA Tesla P100 vs P40

With regards to speedup, on CuSha with either internal representation, NVIDIA Tesla P100 performed a little bit slower than NVIDIA Tesla P40 on *email-Enron* and *soc-Slashdot0902*, but was almost twice as fast on all the other graphs.

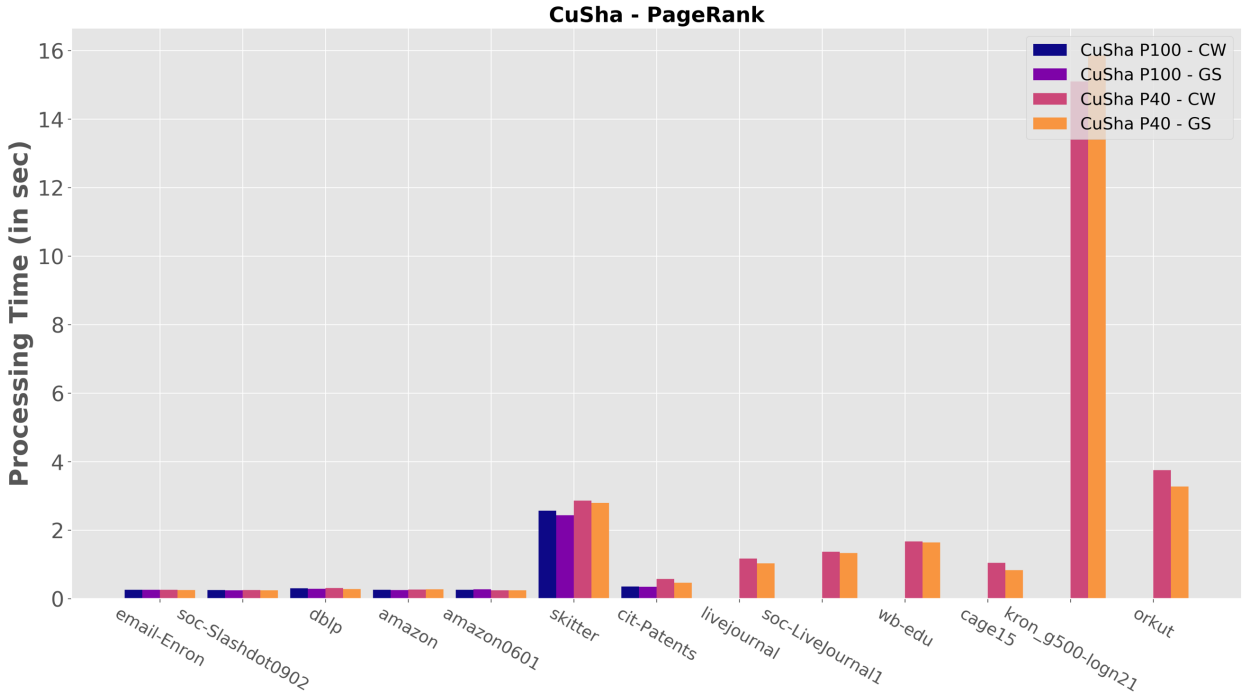

Figure 8: CuSha PageRank processing times on NVIDIA Tesla P100 and P40 GPUs

PageRank on CuSha failed with an out of memory error on some of the larger graphs on the NVIDIA Tesla P100. On the NVIDIA Tesla P40, which has 6GB more of memory than the P100, all experiments did finish successfully but took as long as ~284.6 sec (CW) and ~98.6 sec (GS) on *wikipedia-2007*, and ~668.6 sec (CW) and ~325 sec (GS) on *indochina-2004*. To avoid dwarfing all the other plots, these two graphs have not been depicted in the above graph. Barring *kron_g500-logn21*, *orkut* and *skitter*, processing PageRank on all the other graphs took less than 2 sec. Between CW and GS, once again, GS fared better on most graphs.

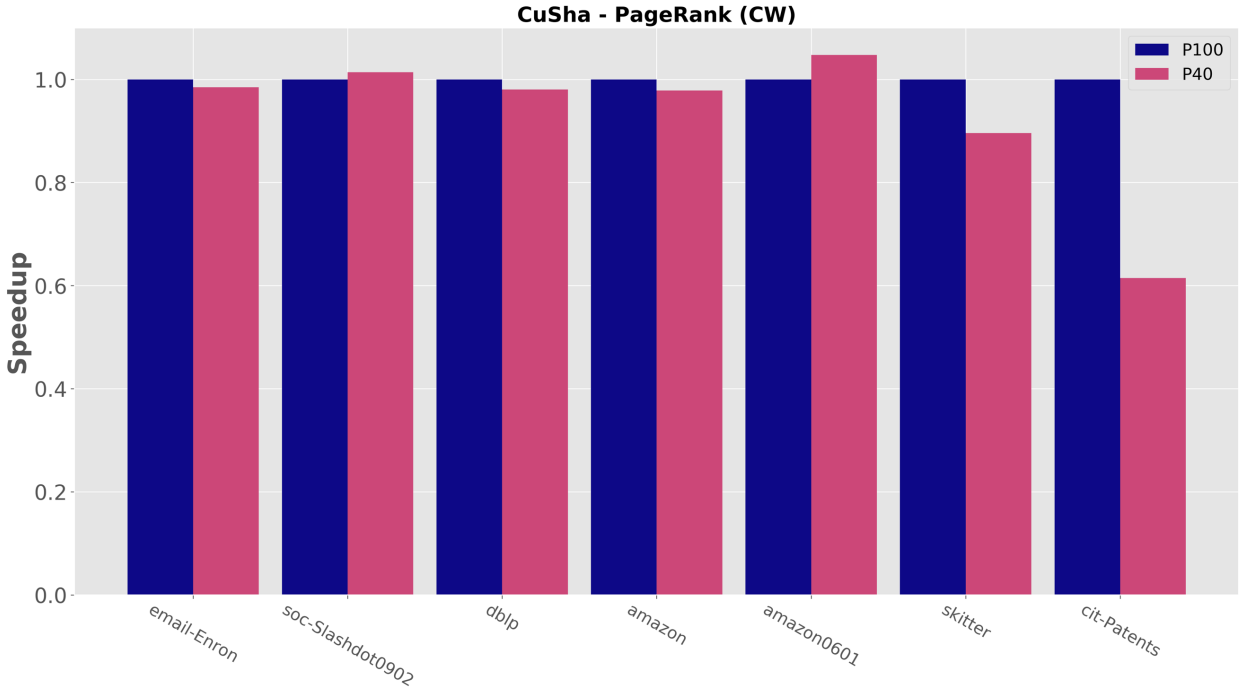

Figure 9: CuSha PageRank (CW) speedup - NVIDIA Tesla P100 vs P40

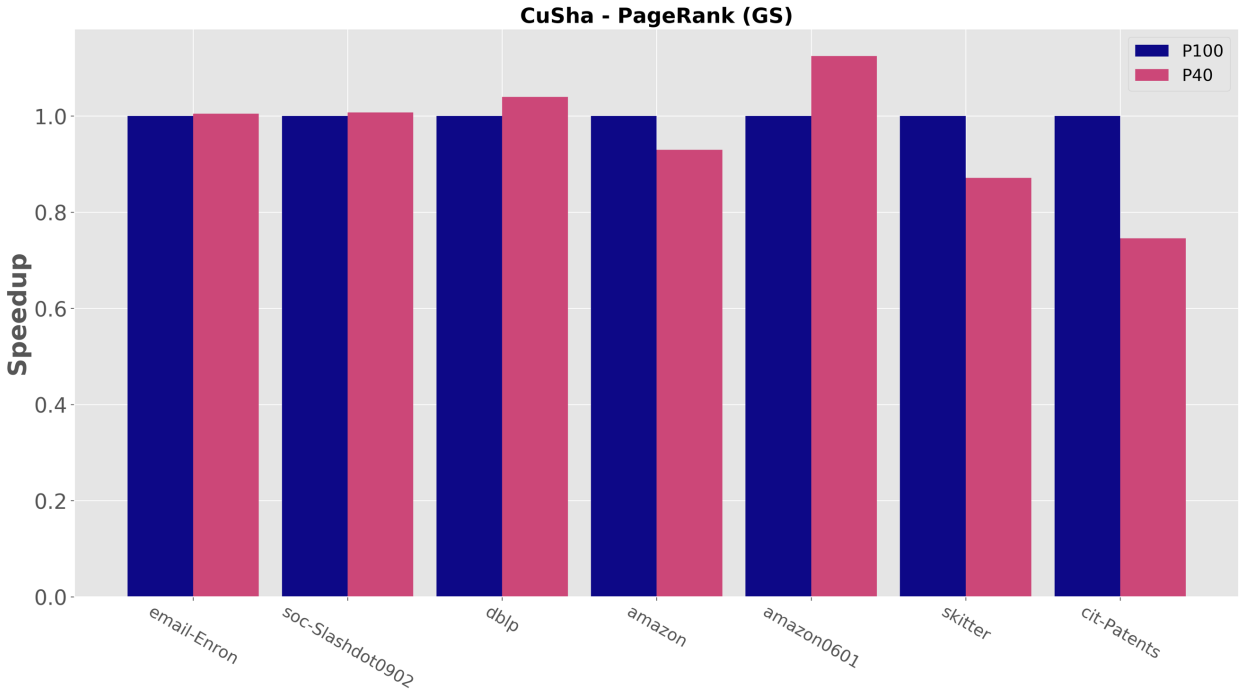

Figure 10: CuSha PageRank (GS) speedup - NVIDIA Tesla P100 vs P40

Since PageRank on CuSha ran out of memory for some of the larger graphs on NVIDIA Tesla P100, the speedup plots were not computed for those graphs. For the ones shown above, excepting *soc-slashdot0902* and *amazon0601* on CW and *dblp* and *amazon0601* on GS, P100 was faster than P40.

3. Gunrock

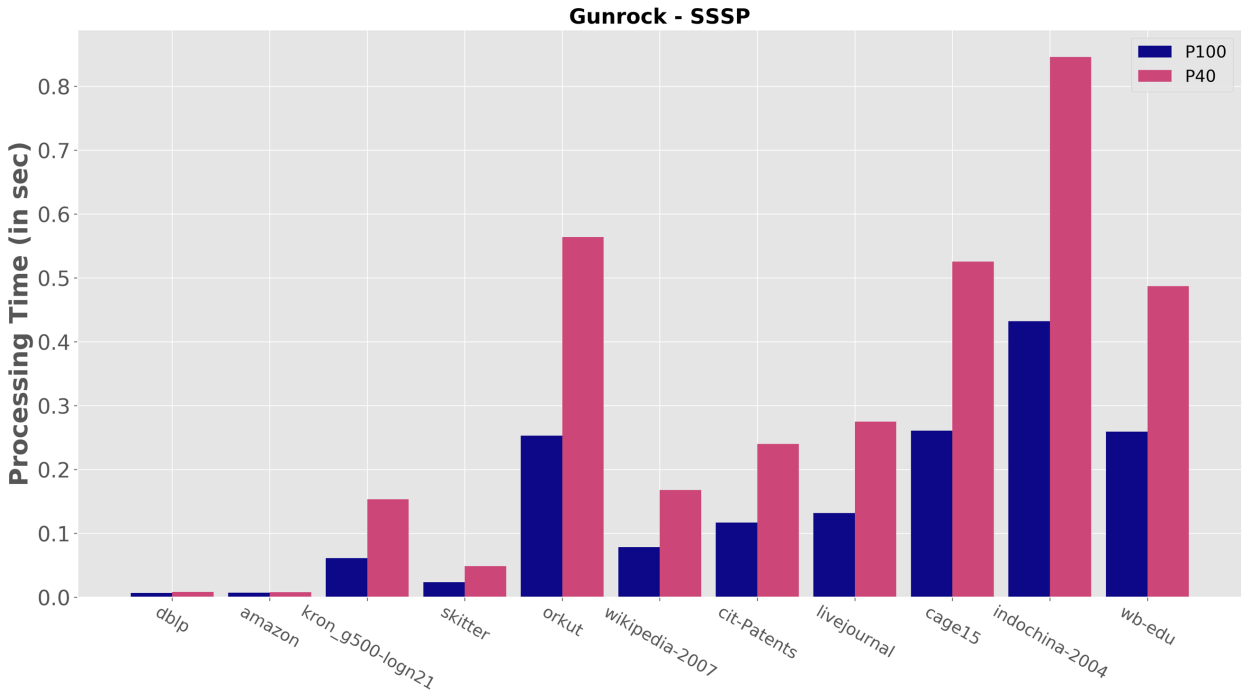

Figure 11: Gunrock SSSP processing times on NVIDIA Tesla P100 and P40 GPUs

Both NVIDIA Tesla P40 and NVIDIA Tesla P100 had similar behavioral patterns when executing SSSP on Guncrock. On both GPUs, processing *indochina-2004* took the longest at ~0.85 sec (P40) and ~0.42 sec (P100). This was followed by *orkut* at ~0.56 sec (P40) and ~0.25 sec (P100), *cage15* at ~0.52 sec (P40) and ~0.26 sec (P100), and *wb-edu* at ~0.49 sec (P40) and ~0.26 sec (P100).

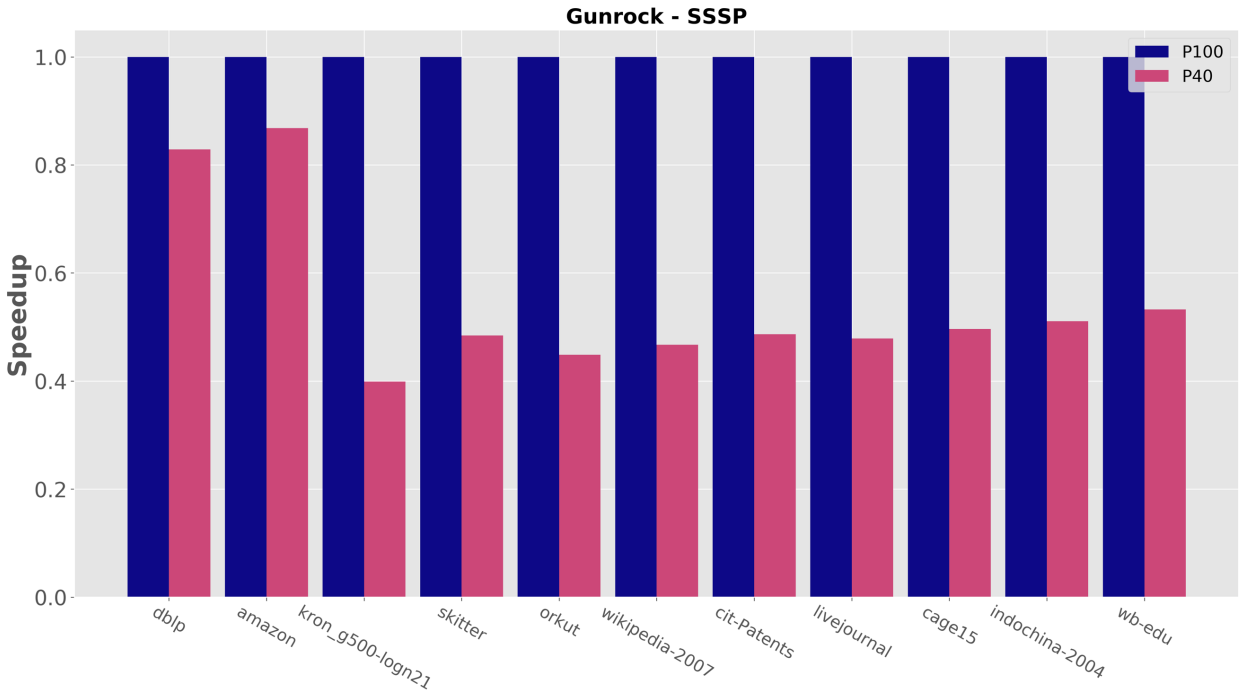

Figure 12: Gunrock SSSP speedup - NVIDIA Tesla P100 vs P40

Between the two GPUs, on all graphs, NVIDIA Tesla P100 performed better than NVIDIA Tesla P40 and once again, almost twice as much on most of them.

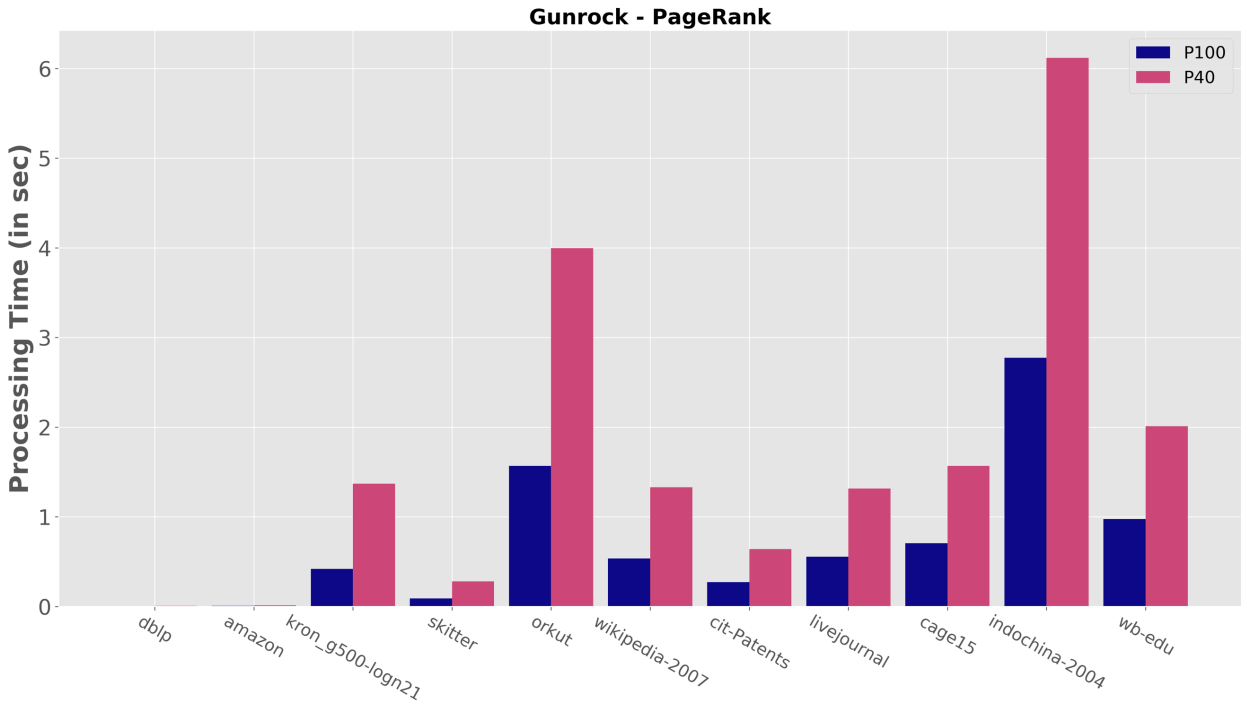

Figure 13: Gunrock PageRank processing times on NVIDIA Tesla P100 and P40 GPUs

For PageRank on Gunrock, both GPUs exhibited similar trends as well. Processing *indochina-2004* took the most time at ~6.1 sec (P40) and ~2.8 sec (P100), followed by *orkut* at ~4 sec (P40) and ~1.6 sec (P100). All the other graph topologies took less than or equal to ~2 sec on both GPUs.

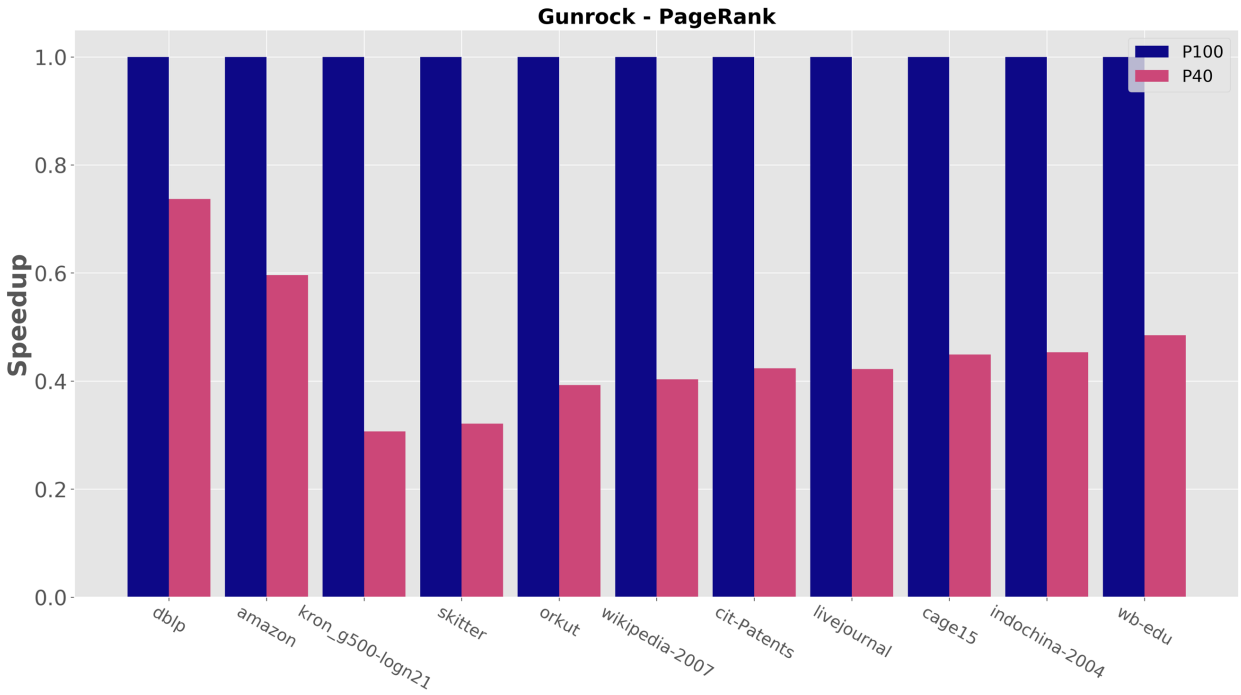

Figure 14: Gunrock PageRank speedup - NVIDIA Tesla P100 vs P40

Once again, between the two GPUs, NVIDIA Tesla P100 was consistently much faster, more than twice as much on most graphs.

Discussion

Of the two GPUs considered, NVIDIA Tesla P100 yielded better results than NVIDIA Tesla P40 on all frameworks and most graphs. The difference in performance wasn't significant for smaller graphs, but for some of the mid- and large-sized graphs, P100 was almost twice as fast as P40. The speedup was likely a result of one or both of higher frequency/more cores on the P100 and its memory subsystem, but deeper analysis would require profiling using NVIDIA's profiling tool *nvprof*.

Even though P100 performed better than P40 overall, we were unable to obtain results for all graphs on it for CuSha PageRank due to its memory limitations. Hence, for the graphs in this section, we only consider the results from P40. However, it must be noted that the results from P100 would have also resulted in similar-looking graphs because of identical behavioral trends on both GPUs, as noted in the previous section.

Figure 15 shows the processing times for all frameworks on SSSP on NVIDIA Tesla P40. Across all graphs, CuSha with either representation took the least time followed by Gunrock and then nvGRAPH. Within CuSha, the performance with G-Shards was consistently slightly better than with Concatenated Windows.

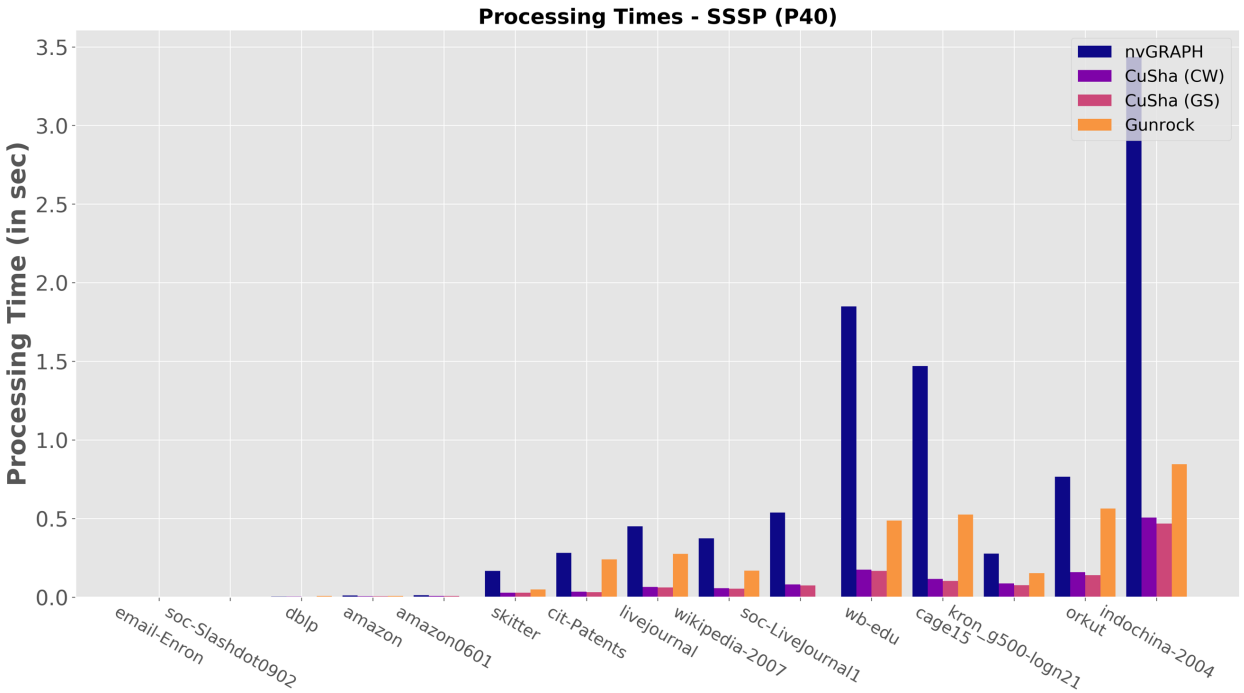

Figure 15: Processing times - SSSP for nvGRAPH, CuSha (CW), CuSha (GS), Gunrock on NVIDIA Tesla P40

- Both Bulk-Synchronous Parallel (BSP) frameworks yielded good performance on SSSP, a traversal-based algorithm. With CuSha, the bulk-synchronous steps were split into *Gather*, *Apply*, and *Scatter*, whereas with Gunrock, the steps comprised of *Advance*, *Filter* and *Compute* phases. In both frameworks, different super-steps had dependencies between them but individual operations within a step could be processed in parallel. For example, the accumulation of information about adjacent vertices in an active vertex, or the computation on vertices in a frontier, or the identification of neighboring vertices to form the next active frontier, could all be parallelized across vertices [8]. In general, our results reaffirmed that BSP operations are well-suited to efficient implementation on the GPU because they provide enough parallelism to avoid GPU underutilization and at the same time do not require expensive fine-grained synchronization or locking operations [8].

- Executing SSSP on smaller graphs took comparable times on CuSha and Gunrock but on larger graphs, CuSha was much faster. This is because by organizing graphs into G-Shards and Concatenated Windows, CuSha was able to coalesce memory accesses, avoid intra-warp path divergence and enhance GPU utilization, resulting in better overall workload distribution and load-balancing than Gunrock. Neither CuSha nor Gunrock appeared to be sensitive to the diameters of the graphs considered.
- Between CuSha's internal representations, the results with G-Shards were consistently slightly better than with Concatenated Windows. This was likely because in the case of Concatenated Windows, even though the number of memory transactions were the same as with G-Shards, the memory accesses were not always fully coalesced. Besides, with just G-Shards, the abundance of shards to be processed was already enough to keep Streaming Multiprocessors (SMs) busy, and the introduction of Concatenated Windows didn't help increase GPU utilization any further.
- nvGRAPH was observed to be the much slower than CuSha and Gunrock on all graphs, although the difference was unnoticeable on smaller graphs. Unlike the other frameworks, however, nvGRAPH was more sensitive to the diameter of graphs; it took much longer than CuSha and Gunrock to process three of the scale-free low-diameter graphs: *wb-edu*, *cage15* and *indochina-2004*.

Figure 16 depicts the performance of executing PageRank on all three frameworks across all graphs other than *wikipedia-2007* and *indochina-2004* on NVIDIA Tesla P40; these two plots were omitted since CuSha took the longest at ~284.6 sec (CW) and ~98.6 sec (GS) on *wikipedia-2007*, and ~668.6 sec (CW) and ~325 sec (GS) on *indochina-2004*, and including these plots dwarfed all the other ones. In general, for processing PageRank, a dense-computation-based primitive, there was not a single best framework. Instead, both nvGRAPH and Gunrock appeared equally good for small graphs, nvGRAPH appeared best for larger graphs with low diameters, and Gunrock appeared best for larger graphs with high diameters.

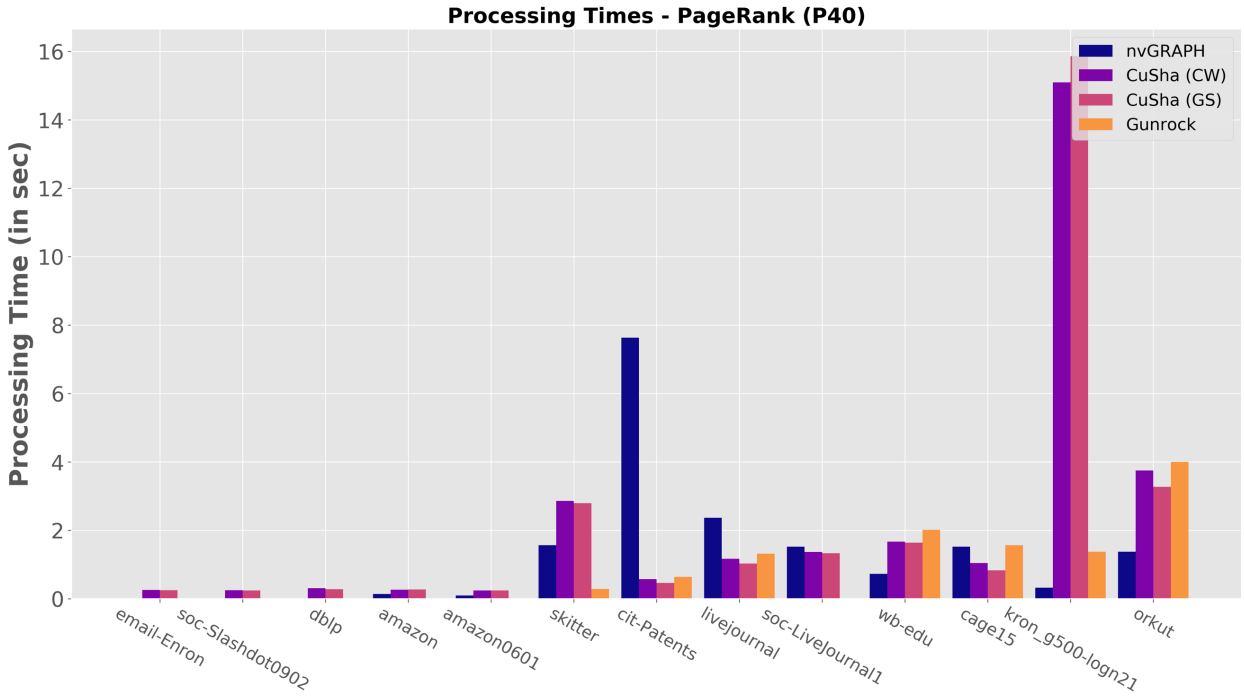

Figure 16: Processing times - PageRank for nvGRAPH, CuSha (CW), CuSha (GS), Gunrock on NVIDIA Tesla P40

- For small-sized graphs such as *email-Enron*, *soc-Slashdot0902*, *dblp*, *amazon*, and *amazon0601*, irrespective of the diameter, both nvGRAPH and Gunrock performed equally well and better than CuSha.
- For mid- and large-sized graphs with low diameters such as *orkut*, *kron_g500-logn21*, *wb-edu*, nvGRAPH performed better than both CuSha and Gunrock. However, for mid- and large-sized graphs with high diameters such as *cit-Patents*, *liveJournal*, *soc-Livejournal1*, nvGRAPH performed worse. Since nvGRAPH is close-sourced, we were unable to explain why that is so, although profiling with *nvprof* would likely reveal useful insights.
- For mid- and large-sized graphs with high diameters, Gunrock appeared to be the framework of choice. Cusha with G-Shards representation was slightly faster than Gunrock on certain graphs, but also took astronomically longer than both Gunrock and nvGRAPH on *wikipedia-2007* and *indochina-2004*. In terms of memory usage, CuSha was also much more memory-intensive, failing with out-of-memory errors on some of the larger graphs on P100. Gunrock, on the contrary, did not encounter any such problems. This was likely because PageRank is a dense-computation-based primitive in which all vertices are active in a frontier at any given point in time, and clustering vertices into G-Shards and Concatenated Windows in very large graphs in the case of CuSha did not help reduce non-coalesced accesses or improve workload distribution. Instead, Gunrock's dynamic load-balancing optimizations involving course- and fine-grained load-balancing techniques proved much more beneficial.

Conclusion

High-level GPU graph frameworks allow programmers to execute a wide variety of graph algorithms using out-of-the-box graph primitives. These frameworks are built around unique programming models, which define how these frameworks model and approach graph problems. In this paper, we considered three such frameworks – nvGRAPH based on the linear algebra model, CuSha based on the Gather-Apply-Scatter (GAS) model, and Gunrock based on the data-centric model – and studied their performance impacts on two graph algorithms – SSSP and PageRank – on a variety of structurally different graphs and two GPUs – NVIDIA Tesla P40 and NVIDIA Tesla P100.

Through the experiments performed, we found BSP frameworks to be better suited to traversal-based algorithms such as SSSP, due to their ability to efficiently regularize workloads and keep the GPU busy. Amongst the BSP frameworks considered, CuSha with G-Shards representation resulted in the best overall performance. The results from the experiments on PageRank did not imply a single best framework. Instead, 1) for small-sized graphs with any diameter, both nvGRAPH and Gunrock exhibited good performance, and 2) for mid- and large-sized graphs with low diameters, nvGRAPH appeared best, and 3) for mid- and large-sized graphs with high diameters, Gunrock appeared best overall. While CuSha was the fastest on some of the graphs, it was incredibly slower on some of the others, and also occupied the most memory; it failed on some of the larger graphs on the P100 GPU with out-of-memory errors. Between the GPUs, even though the focus of this paper was not on comparing their differences, NVIDIA Tesla

P100 executed algorithms much faster than NVIDIA Tesla P40. The speedup was almost twice as much for most of the mid- and large-sized graphs.

From executing the above experiments, it is evident that selecting the right programming model for a specific graph topology is crucial to realizing the best possible performance on a graph algorithm. Besides, from an architecture standpoint, better hardware, more memory, and system support for load-balancing irregular workloads are also boosters of performance.

Future Work

For future work, it will be interesting to consider asynchronous execution frameworks such as Frog [10] since unlike the frameworks considered in this paper, they won't have to bear the cost of carrier synchronization between super-steps, which can be very expensive. Another potential area for study would be to look at automatic kernel fusion, an optimization that would reduce synchronization costs in GPUs but which isn't performed automatically in current frameworks [9]. Moreover, since we noticed a difference in performance on NVIDIA Tesla P100 and P40 GPUs, it would be useful to further investigate if that difference is a result of more cores, or memory subsystems, or both. This could be done using *nvprof* to see how long specific computations and memory access calls take on each GPU. *nvprof* profiling could also be used to better characterize nvGRAPH without access to its code and help determine why its performance is better than both CuSha and Gunrock on large low-diameter graphs. Lastly, memory performance is of extreme importance in all GPU programming frameworks. Hence, it will be worthwhile to closely evaluate the relationship between the frameworks' performances and the GPUs' memory subsystems, as well as to explore graph formats other than COO or CSC, such as NetflixGraph [11], since they might reduce memory footprint and result in better overall performance.

References

[1] Davidson, Andrew, Sean Baxter, Michael Garland, and John D. Owens. "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths." *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (2014): n. pag. Web.

[2] Keckler, Stephen W., William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. "GPUs and the Future of Parallel Computing." *IEEE Micro* 31.5 (2011): 7-17. Web.

[3] Khorasani, Farzad, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. "CuSha: Vertex-Centric Graph Processing on GPUs." N.p., 2012. Web.

[4] Merrill, Duane, Michael Garland, and Andrew Grimshaw. "Scalable GPU Graph Traversal." *ACM SIGPLAN Notices* 47.8 (2012): 117. Web.

[5] "NvGRAPH." *NVIDIA Developer Documentation*. N.p., n.d. Web.

[6] "SNAP for C++: Stanford Network Analysis Platform." *SNAP: Stanford Network Analysis Project*. N.p., n.d. Web.

[7] "SuiteSparse Matrix Collection Formerly the University of Florida Sparse Matrix Collection." *SuiteSparse Matrix Collection*. N.p., n.d. Web.

[8] Wang, Yangzihao, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. "Gunrock: A High-performance Graph Processing Library on the GPU." *ACM SIGPLAN Notices* 50.8 (2015): 265-66. Web.

[9] Wu, Yuduo, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens. "Performance Characterization of High-Level Programming Models for GPU Graph Analytics." *2015 IEEE International Symposium on Workload Characterization* (2015): n. pag. Web.

[10] Shi, Xuanhua, et al. “Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model.” Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015, 2015, doi:10.1145/2688500.2688542.

[11] Netflix. “Netflix/Netflix-Graph.” GitHub, 17 Nov. 2017, github.com/Netflix/netflix-graph.

[12] Brin, Sergey, and Lawrence Page. “The Anatomy of a Large-Scale Hypertextual Web Search Engine.” Computer Networks and ISDN Systems, vol. 30, no. 1-7, 1998, pp. 107–117., doi:10.1016/s0169-7552(98)00110-x.

[13] “NVIDIA Tesla P40 GPU Accelerator.” NVIDIA, images.nvidia.com/content/pdf/grid/data-sheet/nvidia-p40-datasheet.pdf.

[14] “NVIDIA Tesla P100 GPU Accelerator.” NVIDIA, images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.